**红黑树插入算法实验报告**

1. **实验目的**

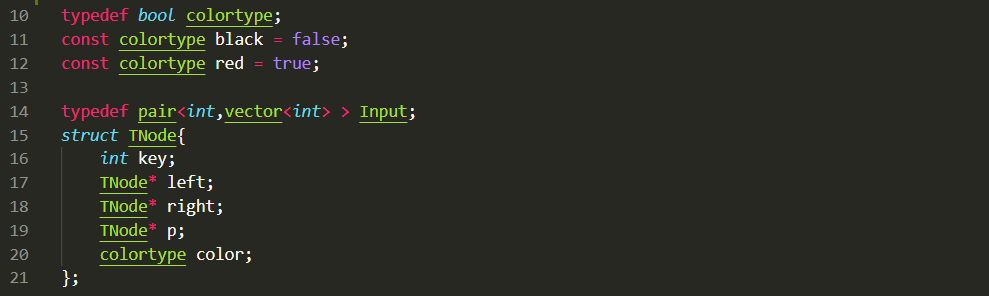
通过实践，加深理解红黑树的性质、特点，熟悉其插入旋转等相关操作，提高编程能力。

1. **实验内容**
2. 编码实现红黑树的插入算法，使得插入后依旧保持红黑性质，即实现RB\_INSERT, RB\_INSERT\_FIXUP算法。
3. 从insert.txt文件中按照指定格式读取数据并构建一棵红黑树，再对这棵红黑树进行层次遍历，将遍历结果按照指定格式输出到LOT.txt文件中。
4. **实验步骤**
5. **定义红黑树的数据结构：**

我将红黑树的数据结构定义为一个两层结构：红黑树的节点和树。

1. **红黑树节点**

将红黑树节点定义为一个struct，命名为TNode，其内部包含五个字段：分别为key，left指针，right指针，p指针以及color。



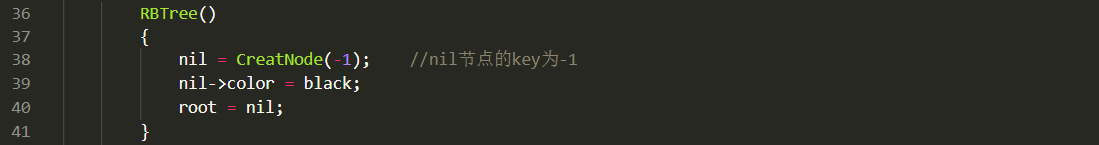
1. **红黑树**

将红黑树的树结构定义为一个class，命名为RBTree，其私有属性包括一个nil指针和一个root指针，分别指向红黑树的nil节点和根节点。另外，RBTree中还声明了一个默认构造函数RBTree()，四个友元函数RB\_Insert()、RB\_Insert\_Fixup()、left\_rotate()和right\_rotate()以及Nil()和Root()。



1. **构造函数RBTree()**

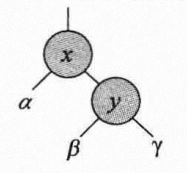
它用于对一棵红黑树进行初始化。首先它会调用CreatNode()函数来生成一个nil节点，可以看到，在这里我将nil节点的关键字设为-1以与其他内部结点作区分，同时将nil节点的左右孩子以及父亲指针均置为空，由于新生成的节点颜色均为red，因此在构造函数中要显式地将nil节点的颜色置为black，最后将root指针指向这个nil节点，此时红黑树的初始化就完成了。





1. **left\_rotate()和right\_rotate()**

right\_rotate()所做的操作与left\_rotate()完全对称，因此，这里只介绍left\_rotate()是如何设计的：



对上图中x节点进行左旋操作，首先令指针y指向x的右孩子，再将y的左孩子链到到x的右孩子指针上，如果y的左孩子不为空，还要将y的左孩子的父指针链到x上，再将y的父指针链到x的父指针上，如果x的父指针指向nil节点，说明x是root节点，还要将此红黑树的根节点设为y(即令root指针指向y)，如果x不根节点，那么x是其父节点的左孩子或右孩子，若是左孩子，将x的父节点的左孩子指针指向y，否则将右孩子指针指向y，再将y的左孩子指针指向x，最后令x的父指针指向y，并返回这棵旋转过后的红黑树。



1. **RB\_Insert()**

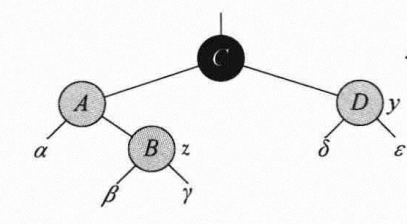
红黑树的插入操作与二叉树极其类似，只不过红黑树在插入操作之后还要进行一步FixUp操作，以维护它的红黑性质。在向红黑树T插入一节点z时，首先要寻找插入位置，先令指针y指向nil节点，再令x指向root节点，再执行while循环：令y指针指向x（为叙述方便，后面指针指向的节点均以指针代替），如果待插入节点key小于x的key，那么往左子树上找插入位置，否则往右子树上找插入位置。当退出while循环时，y指针指向插入位置的父节点，故令z的父指针指向y，如果此时y(插入位置的父节点)是nil节点，说明插入位置是根节点的位置，故要将root指针指向z，若插入位置是其父节点的左孩子，则将y的左孩子指向z，否则将y的右孩子指向z。接下来将插入结点的左右孩子指向nil节点，并将其颜色设为red。最后执行FixUp操作并返回插入结点并调整之后的树T。



1. **RB\_Insert\_Fixup()**

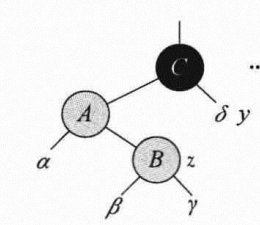
在执行FixUp操作时，分为6种情况,根据插入结点z的父节点是z的祖父节点的左孩子还是右孩子又分为前三种和后三种情况两大类，其中case 4、case 5、case 6与前三种情况对称，故这里只讨论前三种情况：

Case1：



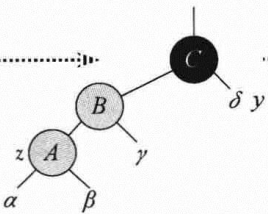
如上图所示，插入结点z的父亲和叔叔y均是红结点，将z的父亲和叔叔y的颜色均置为black，再将z的祖父节点的颜色置为red。case1执行结束之后可能还会转入case2 或case3。

Case2：



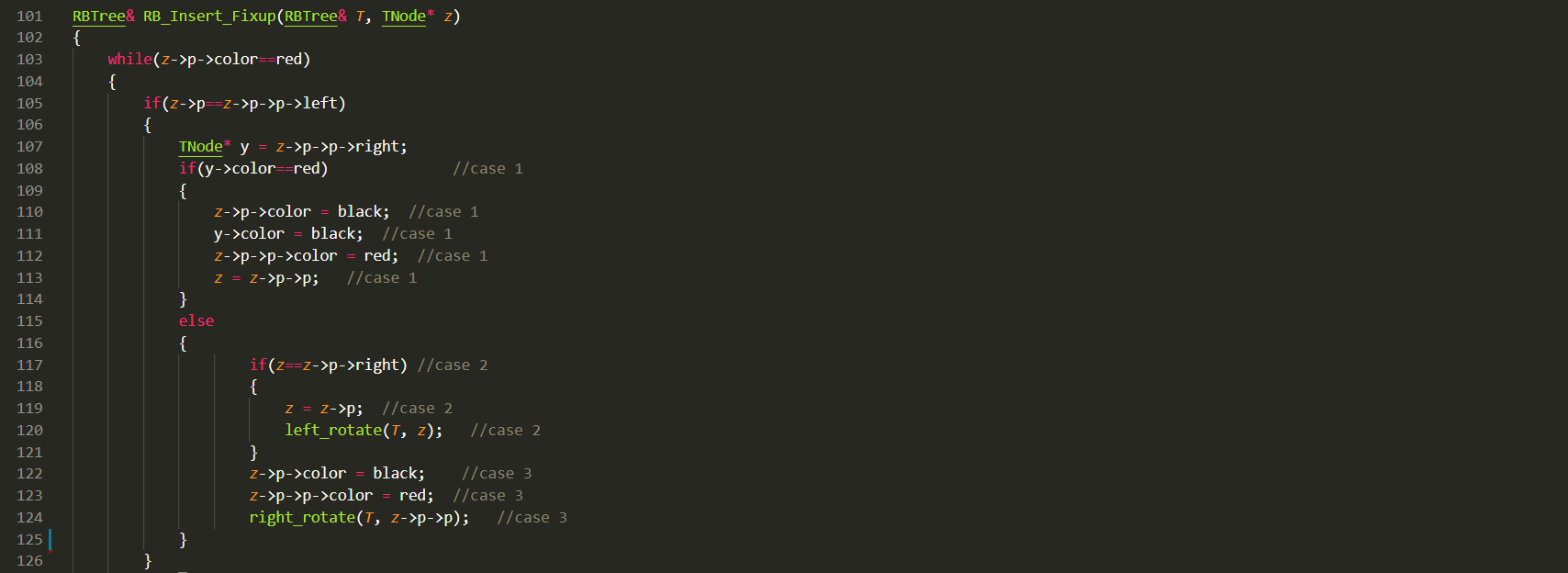
如上图所示，当z的父节点为red，但叔叔y为black，且z为其父节点的右孩子时属于case2，首先将z指针指向它的父节点，再对新的z指针指向的节点执行左旋操作，并转入对case2的处理。

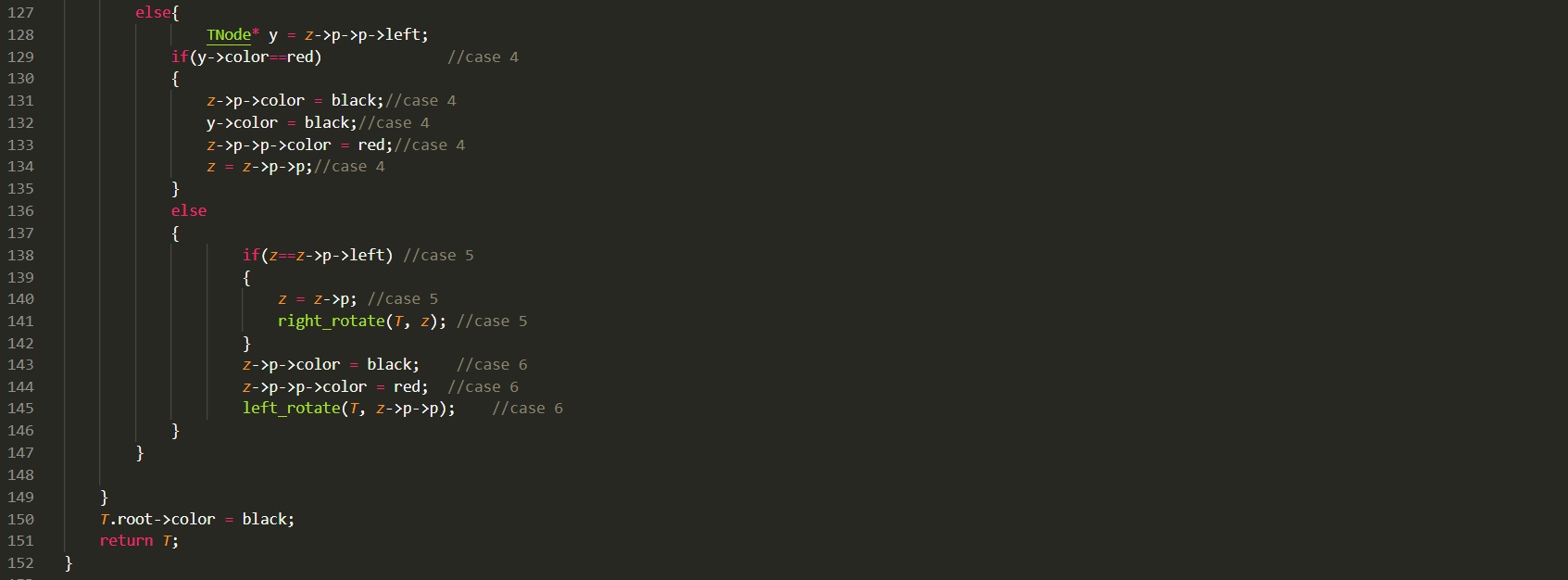
Case3：



如上图所示，插入结点z和其父亲均为red，其叔叔y为黑色，首先z的父亲变为black，再将z的祖父变为red，接着对z的祖父执行右旋操作。

在完成一次调整之后，指针z会在树中上移，此时如果z指向的新节点和其父亲均为红色，此时还要继续执行上面的步骤，直到z的父亲节点为black，退出这个循环调整过程。Fixup的最后一步操作是将根节点的颜色置为black，之所以要这样做是因为可能出现这样两种情况：①插入的节点为树中的第一个节点，②指针z上溯到根节点，这两种情况中z的父节点均为black，但是root节点为red，因此我们加上一步将root节点调黑的操作。最后返回调整之后的红黑树。





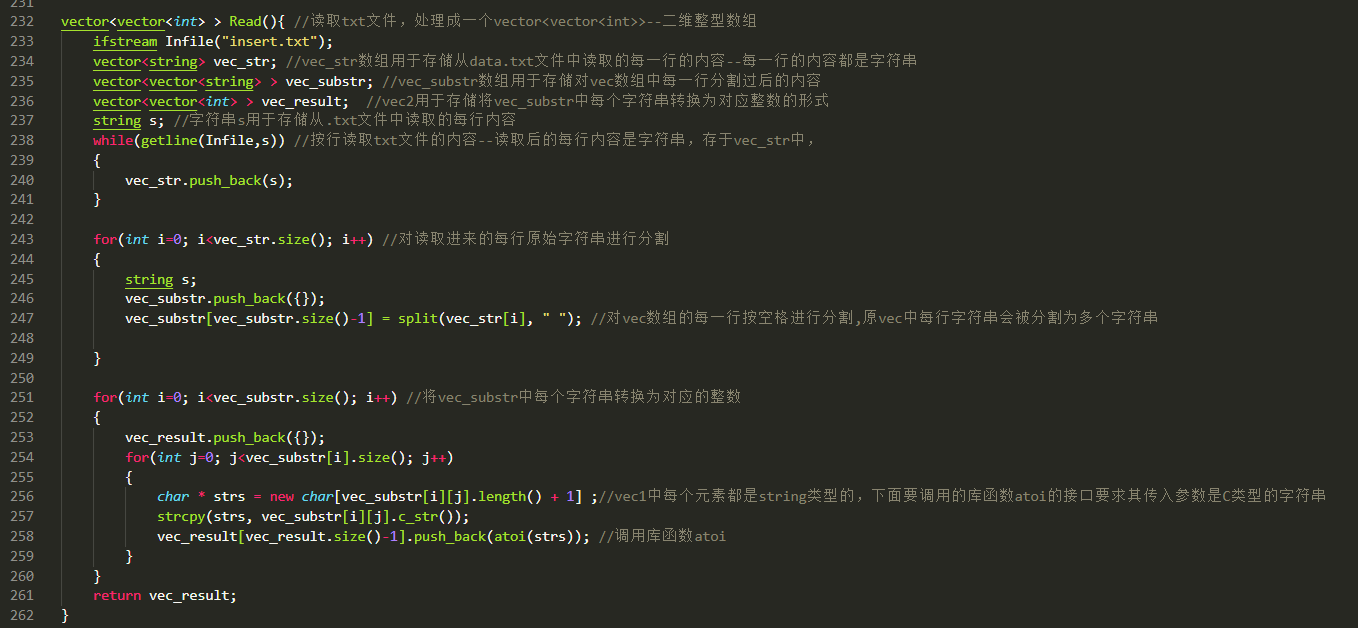
1. **输入**

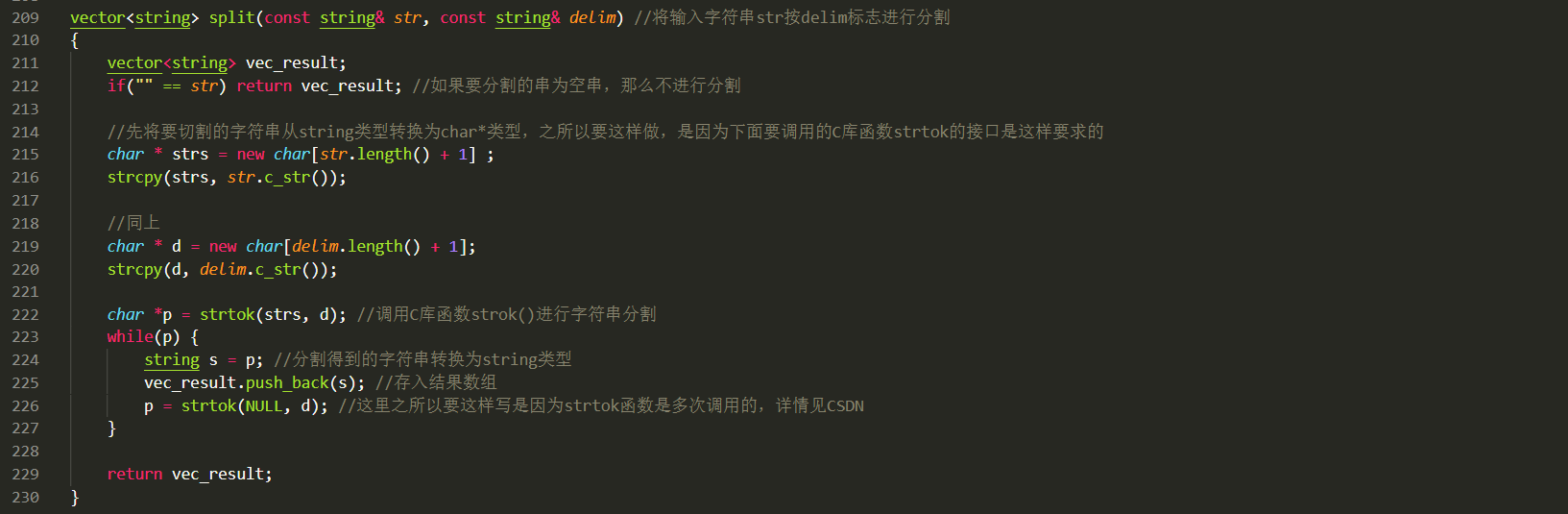
设计Read()函数读取输入数据，由于输入数据分两行保存在.txt文件中，首先调用函数getline()按行读取数据，读取到的每行数据都是一行字符串。接下来又设计了split()函数，用于实现对字符串进行分割，其中split()函数的分割功能是通过调用C库函数strtok()实现的。

Strtok()函数的调用格式如下所示：

可以通过对strtok( )函数的一连串调用来将输入字符串str分割为一个个token—一串由delimiters字符串中的某个子串分割出来的连续字符。在第一次调用strtok()函数时，需要给strtok()函数的第一个参数传入待分割的字符串str，它的第一个字符会作为寻找token的扫描起点，当扫描出一个token时，返回这个token的开始位置。在接下来的调用中，strtok函数的第一个参数均传入null指针，函数会从上个token的结束位置的下个位置开始扫描。当扫描到str字符串的结束标志null字符时停止分割。此时，在接下来的对strtok函数的调用中，返回的均是null指针。

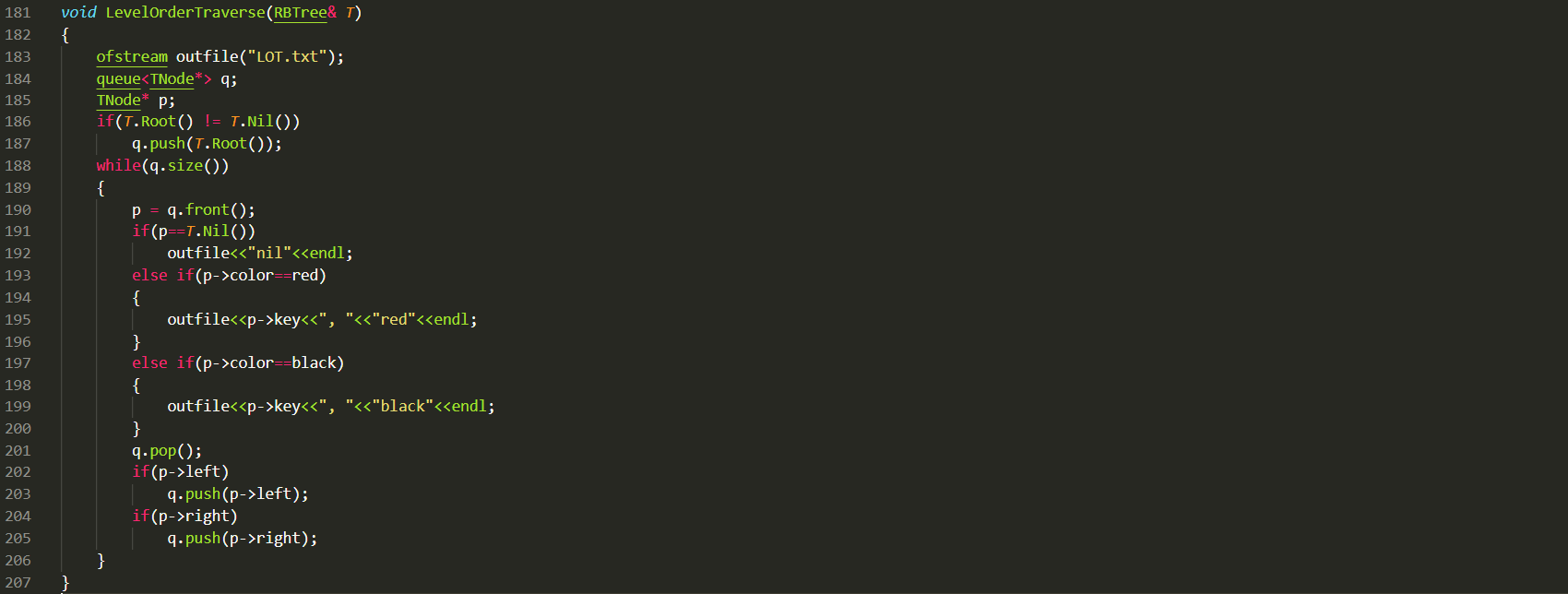
实现完字符串分割以后，再通过调用C库函数atoi()将每个待输入的以字符串表示的关键字转换成对应的整数形式，此时这些数据就可以用于构建一棵红黑树了。





1. **层次遍历及输出**

采用层次遍历来搜索前面建立好的红黑树，首先新建一个LOT.txt文件用于输出，在层次遍历过程中，每次遍历到一个节点首先要判断它是nil节点还是内部结点，如果是nil节点就输出“nil”，并换行继续输出，如果是内部结点，就输出节点的key值以及节点颜色，并换行继续输出。

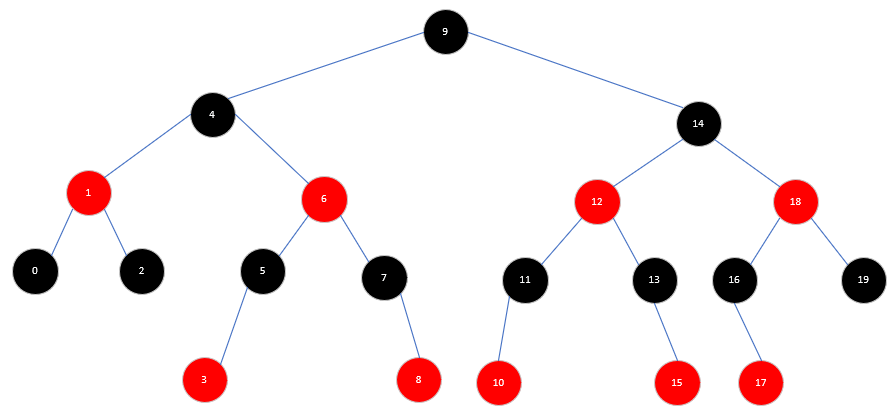
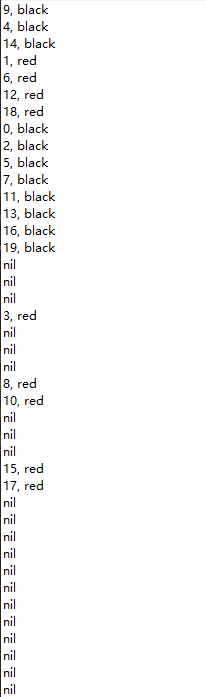


1. **实验结果**

对于以下输入：



建立好红黑树并采用层次遍历的输出以及该红黑树可视化形式为：



**五、实验心得**

在编写本实验源码的过程中，无论是输入输出数据还是插入算法的实现均比较复杂，于是我采用这样的思路进行编写源码：**1.**我将源码划分为两大部分：插入算法实现，以及IO操作，首先实现插入算法部分，接着再实现IO操作部分。**2.**在编写源码过程中，无论是实现插入操作还是IO，总是先确定好自己为实现这一部分的功能需要定义的函数及函数的接口，包括传入参数、传出参数以及返回值的类型等，接下来再根据算法的具体过程来细化各个函数的内容。**3.**在实现插入算法过程中，我是用insert()函数调用FixUp()函数，再用FixUp()函数调用rotate()函数，按照insert->FixUp->rotate这样的顺序来定义函数接口，再按照相反地顺序来实现各个函数的内部细节。按照上述思路我在编写源码的整个过程中思路都很清晰，以后在编写代码量比较大的源码时也会按照这样的思路进行编写。